NetworkX Quick Reference

More detailed documentation and listing of options and defaults can be found in the html documentation or by using pydoc (or interactive help) on a function, method or class. For example, for methods of the Graph class such as add_node, use

```
pydoc networkx.Graph.add_node
```

or

```
pydoc networkx.Graph
```

to report all Graph methods.

For functions associated with multiple graph classes, such as subgraph or watts_strogatz_graph, use

```
pydoc networkx.subgraph
```

or

```
pydoc networkx.watts_strogatz_graph
```

Terminology

Graph or network structure is encoded in the **edges** (connections, links, ties, arcs, bonds) between **nodes** (vertices, sites, actors).

```
nlist     - a list of nodes.
nbunch    - a bunch of nodes:
            any iterable container of nodes.
e=(u,v)   - a Graph or DiGraph edge as a Python tuple (also written u-v or u->v).
e=(u,v,x) - an XGraph or XGraph edge as a Python tuple, x is an arbitrary object.
elist     - a list of edges.
ebunch    - a bunch of edges:
            any iterable container of edge-tuples.
```

Creation

```
G=Graph()      - create empty simple graph G.
G=DiGraph()    - create empty simple directed graph G.
G=XGraph()     - create empty graph G with edge data.
G=XDiGraph()   - create empty directed graph G with edge data.
G=empty_graph(n) - create empty graph with n nodes.
G=empty_graph(n,create_using=DiGraph()) - create empty digraph with  n nodes.
G=create_empty_copy(H) - create new, empty graph of same class as H.
```

Manipulation

Methods associated with a graph-like object G:

```
G.add_node(n)                 - add single node to G.
G.add_nodes_from(nbunch)      - add each node in nbunch to G.
G.delete_node(n)              - delete node n from G.
G.delete_nodes_from(nbunch)   - delete each node n in nbunch.
G.add_edge(u,v)               - add edge (u,v) to G.
                                if G is a digraph, add directed edge u->v.
G.add_edge(e)                 - add edge e=(u,v) *(equivalent to above)*
G.add_edges_from(ebunch)      - add each edge e in ebunch to G.
G.delete_edge(u,v)            - delete edge (u,v)
G.delete_edge(e)              - delete edge e=(u,v)
G.delete_edges_from(ebunch)   - delete each edge in ebunch from G.

G.add_path(nlist)             - add nodes and edges to make ordered path.
G.add_cycle(nlist)            - same as add_path, but end nodes are
                                connected.
G.clear()                     - delete all nodes and edges.
G.copy()                      - return "shallow" copy of the graph
                                (like dict.copy())
G.subgraph(nbunch)            - return subgraph induced by nodes in nbunch.
```

New graphs from old

```
subgraph(G, nbunch)                 - subgraph induced by nodes in nbunch.
union(G1,G2)                        - graph union.
disjoint_union(G1,G2)               - graph union, assuming all nodes are different.
cartesian_product(G1,G2)            - Cartesian product graph.
compose(G1,G2)                      - combine graphs, identifying nodes with same name
complement(G)                       - return graph complement.
create_empty_copy(G)                - empty copy of the same graph class.
convert_to_undirected(G)            - return an undirected copy of G.
convert_to_directed(G)              - return a directed copy of G.
convert_node_labels_to_integers(G) - return copy with nodes relabed as integers.
```

Graph Properties

Methods:

```
G.order()            - number of nodes in G.
G.size()             - number of edges in G.
```

```
G.nodes()            - return copy of all nodes of G in a list.
G.nodes_iter()       - return iterator over all nodes in G.
G.has_node(n)        - True if n is a node in G.
n in G               - equivalent to G.has_node(n)
G.edges()            - return list of all edges in G.
G.edges(nbunch)      - return list of edges adjacent to some node in nbunch.
G.edges_iter()       - return iterator over all edges in G.
G.edges_iter(nbunch) - return iterator that iterate once over
                       each edge adjacent to some node in nbunch.
G.has_edge(u,v)      - True if (u,v) is an edge in G.
G.edge_boundary(nb1) - return a list of edges outward from nb1.
G.edge_boundary(nb1,nb2) - return a list of edges between nb1 and nb2.

G.neighbors(n)       - return list of nodes connected to node n (outgoing if directed
G[n]                 - equivalent to G.neighbors(n).
G.neighbors_iter(n)  - return iterator over the neighbors of node n.
G.has_neighbor(v,u)  - check if u is a neighbor of v (returns True or False).
G.node_boundary(nb1) - return list of nodes outside but connected to nb1.
G.node_boundary(nb1,nb2) - return list of nodes in nb2 connected to nb1.

G.degree(n)                 - return degree of node n.
G.degree()                  - return list of degrees of all nodes in G.
G.degree(with_labels=True) - return dict mapping each node in G to
                             its degree.
G.degree(nbunch)            - return list of degrees of all nodes in nbunch.
G.degree(nbunch,with_labels=True) - return dict mapping each n in nbunch to degree(n)
```

Directed Graphs Only

```
G.out_edges()           - like edges, but only outward pointing edges.
G.in_edges()            - like edges, but only inward pointing edges.
G.in_degree()           - like degree, but only inward edges count.
G.out_degree()          - like degree, but only outward edges count.
G.predecessors()        - like neighbors, but only inward edges count.
G.successors()          - like neighbors, but only outward edges count.
G.predecessors_iter() - like neighbors_iter, but only inward edges count.
G.successors_iter()     - like neighbors_iter, but only outward edges count.
```

Functions

```
number_of_nodes(G) - number of nodes in G.
order(G)           - equivalent to above.
number_of_edges(G) - number of edges in G.
```

```
size(G)            - equivalent to above.
density(G)         - fraction of possible edges which exist.

nodes(G)      - return copy of all nodes of G in a list.
nodes_iter(G) - return iterator over all nodes in G.
edges(G)      - return list of all edges in G.
edges_iter(G) - return iterator over all edges in G.

diameter(G)  - return maximum of all-pairs shortest path.
periphery(G) - return list of nodes with eccentricity equal to diameter.
radius(G)    - return minimum of all-pairs shortest path.
center(G)    - return list of nodes with eccentricity equal to radius.

is_directed(G)                  - True if G is a directed graph.
is_connected(G)                 - True if G is a connected graph.
number_connected_components(G) - number of connected components in G.
connected_components(G)        - list of lists of nodes in each component of G.
average_clustering(G)          - clustering coefficient averaged over nodes of G.
transitivity(G)                - fraction of transitive triples that are triangles.
communities(G)                 - list of lists storing binary-tree community dendrogr
kl_connected_subgraph(G)       - subgraph of G that is kl-connected.
is_kl_connected(G)             - True if G is kl-connected.

adj_matrix(G)                  - adjacency matrix for G as a numpy matrix.
laplacian(G)                   - Graph Laplacian for G as a numpy matrix
generalized_laplacian(G)       - generalized graph Laplacian for G as a numpy matrix.

is_directed_acyclic_graph(G)   - True if DAG.
topological_sort(G)            - list of nodes in directed graph such that every edge
```

Nodal Properties

If n is unspecified, then report properties of all nodes in graph.

```
neighbors(G,n)                - neighbors (outgoing if directed) of n in G.
G[n]                          - same as above.
degree(G,n)                   - number of edges for n in G.
eccentricity(G,n)             - maximum of shortest-path lengths from n to anywhere i
triangles(G,n)                - number of triangles which include n.
clustering(G,n)               - clustering coefficient: ratio of triangles to potenti
node_betweenness(G,n)         - number of shortest paths through n.
betweenness_centrality(G,n)   - fraction of shortest paths that go through n.
degree_centrality(G,n)        - frction of possible nodes connected to n.
```

```
closeness_centrality(G,n)     - 1/(average distance to all nodes from n).

shortest_path(G,u,v)          - list denoting the shortest path from u to v.
shortest_path_length(G,u,v)   - length of the shortest path from u to v.
node_connected_component(G,n) - list of nodes in node n's connected component.
dijkstra(G,u)                 - dicts for shortest weighted paths and path length fro
dijkstra_shortest_path(G,u)   - dict of paths from u keyed by target node.
dijkstra_path_length(G,u)     - dict of path lengths from u keyed by target node.
```

Generating Graphs

Variable size graphs

```
make_small_graph(graph_description,create_using=None,**kwds)
LCF_graph(n, shift_list, repeats)

balanced_tree(r, h)
barbell_graph(m1, m2)
complete_graph(n)
complete_bipartite_graph(n1, n2)
circular_ladder_graph(n)
cycle_graph(n)
empty_graph(n, create_using=None, **kwds)
grid_graph([m1,m2,...,mk])
grid_2d_graph(m, n)
hypercube_graph(n)
ladder_graph(n)
lollipop_graph(m, n)
null_graph(create_using=None, **kwds)
path_graph(n)
periodic_grid_2d_graph(m, n)
star_graph(n)
wheel_graph(n)
```

Small, named graphs of fixed size

```
bull_graph(), chvatal_graph(), cubical_graph(), desargues_graph(),
diamond_graph(), dodecahedral_graph(), frucht_graph(),
heawood_graph(), house_graph(), house_x_graph(),
icosahedral_graph(), krackhardt_kite_graph(),
moebius_kantor_graph(), octahedral_graph(), pappus_graph(),
petersen_graph(), sedgewick_maze_graph(), tetrahedral_graph(), trivial_graph()
truncated_cube_graph(), truncated_tetrahedron_graph(), tutte_graph()
```

Random graphs

```
barabasi_albert_graph(n, m, seed=None)
binomial_graph(n, p, seed=None)
erdos_renyi_graph(n, p, seed=None)
gnm_random_graph(n, m, seed=None)
gnp_random_graph(n, p, seed=None)
powerlaw_cluster_graph(n, m, p, seed=None)
random_regular_graph(d, n, seed=None)
random_lobster(n, p1, p2, seed=None)
watts_strogatz_graph(n, k, p, seed=None)
```

Graphs from degree sequences

```
configuration_model(deg_sequence, seed=None)
havel_hakimi_graph(deg_sequence, seed=None)
is_valid_degree_sequence(deg_sequence)
create_degree_sequence(n, sfunction=None, max_tries=50, **kwds)
pareto_sequence(n, exponent=1.0)    - return a sequence with pareto distribution of l
powerlaw_sequence(n, exponent=2.0)  - return a sequence with powerlaw distribution of
uniform_sequence(n)                - return a sequence with uniform distribution of l
discrete_sequence(n, distribution)  - return a sequence with distribution matching gi
```

IO

```
read_adjlist(path, comments='#', delimiter=' ', create_using=None, nodetype=None)
write_adjlist(G, path, comments='#', delimiter=' ')
read_edgelist(path, comments="#", delimiter=' ', create_using=None, nodetype=None, ed
write_edgelist(G, path, comments="#", delimiter=' ')
read_multiline_adjlist(path, comments='#', delimiter=' ', create_using=None, nodetype
write_multiline_adjlist(G, path, comments='#', delimiter=' ')
read_gpickle(path)
write_gpickle(G, path)
read_yaml(path)
write_yaml(G, path, default_flow_style=False)
```